

```

/* modi.c

Copyright (c) 1993-2012. Free Software Foundation, Inc.

This file is part of GNU MCSim.

GNU MCSim is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 3
of the License, or (at your option) any later version.

GNU MCSim is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with GNU MCSim; if not, see <http://www.gnu.org/licenses/>

-- Revisions -----
Logfile: %F%
Revision: %I%
Date: %G%
Modtime: %U%
Author: @a
-- SCCS -----

Handles parsing input of the Model Definition File.
*/

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

#include "lexerr.h"
#include "mod.h"
#include "modd.h"
#include "modi.h"
#include "modiSBML.h"

/* Global Keyword Map Structure */

KM vrgkmKeywordMap[] = {
    {"States", KM_STATES, CN_GLOBAL},
    {"Inputs", KM_INPUTS, CN_GLOBAL},
    {"Forcings", KM_INPUTS, CN_GLOBAL}, /* synonym */
    {"Outputs", KM_OUTPUTS, CN_GLOBAL},
    {"Compartments", KM_COMPARTMENTS, CN_GLOBAL},
    {"Dynamics", KM_DYNAMICS, CN_GLOBAL},
    {"Initialize", KM_SCALE, CN_GLOBAL},
}

```

```

    {"Scale",           KM_SCALE,           CN_GLOBAL}, /* obsolete */
    {"Jacobian",        KM_JACOB,          CN_GLOBAL},
    {"Jacob",           KM_JACOB,          CN_GLOBAL}, /* obsolete */
    {"Events",          KM_EVENTS,         CN_GLOBAL}, /* for R deSolve */
    {"Roots",           KM_ROOTS,          CN_GLOBAL}, /* for R deSolve */
    {"CalcOutputs",     KM_CALCOUPUTS,     CN_GLOBAL},

    /* Can be LHS only in Dynamics */
    {"dt",   KM_DXDT,   CN_DYNAMICS | CN_INPUTDEF},

    {"Inline",          KM_INLINE,         CN_ALL},

    {"SBMLModels",      KM_SBMLMODELS,     CN_GLOBAL | CN_TEMPLATE_DEFINED},
    {"PKTemplate",      KM_PKTEMPLATE,     CN_GLOBAL},

    {"End",             KM_END,            CN_GLOBAL | CN_TEMPLATE_DEFINED}, /* Mandatory End
statmt */

    /* Variables names valid in all CN_ */
    {"", 0, CN_ALL} /* This marks the end of map too */

}; /* vrgkmKeywordMap[] */

/* -----
-----
GetKeyword

Returns a pointer to the keyword string given the code. If the
code is not valid, returns empty string.
*/
PSTR GetKeyword (int iCode)
{
    PKM pkm = &vrgkmKeywordMap[0];

    while (*pkm->szKeyword && pkm->iKWCode != iCode)
        pkm++;

    return (pkm->szKeyword); /* Return Keyword Code or null str */

} /* GetKeyword */

/* -----
-----
GetKeyCode

Returns the code of the szKeyword given. If the string is not
a valid keyword or abbreviation, returns 0.
*/
int GetKeyCode (PSTR szKeyword, PINT pfContext)
{

```

```

PKM pkm = &vrgkmKeywordMap[0];

while (*pkm->szKeyword && strcmp (szKeyword, pkm->szKeyword))
    pkm++;

if (pfContext)
    *pfContext = pkm->fContext; /* Set iContext flag */

return (pkm->iKWCode); /* Return Keyword Code or 0 */

} /* GetKeywordCode */

/*
-----
-----
GetVarList

Processes a list of variables (or arrays).
*/
void GetVarList (PINPUTBUF pibIn, PSTR szLex, int iKWCode)
{
    int iLexType, iErr = 0;
    long i, iLB, iUB;
    PSTRLEX szPunct, szTmp;

    do { /* Read model var-list */
        NextLex (pibIn, szLex, &iLexType);

        if (iLexType & LX_IDENTIFIER) { /* identifier found */
            if (GetPunct (pibIn, szPunct, '[')) { /* array */
                GetArrayBounds (pibIn, &iLB, &iUB);
                for (i = iLB; i < iUB; i++) {
                    sprintf (szTmp, "%s_%ld", szLex, i); /* create names */
                    DeclareModelVar (pibIn, szTmp, iKWCode);
                }
            }
            else { /* simple var */
                DeclareModelVar (pibIn, szLex, iKWCode);
            }
        }
        else { /* not an identifier, should be ',' or ')' */
            if ((szLex[0] != ',') && (szLex[0] != CH_RBRACE))
                iErr = szPunct[1] = CH_RBRACE;
        }
    } while ((szLex[0] != CH_RBRACE) && (!iErr));

} /* GetVarList */

/*
-----
-----
ProcessDTStatement

```

```

    process a differential equation definition
 */

void ProcessDTStatement (PINPUTBUF pibIn, PSTR szLex, PSTR szEqn, int
iKWCode)
{
    PSTRLEX szPunct, szTmp;
    PSTREQN szEqnU;
    PINPUTINFO pinfo;
    int iArgType = LX_IDENTIFIER;
    long i, iLB, iUB;

    pinfo = (PINPUTINFO) pibIn->pInfo;

    if (!GetFuncArgs (pibIn, 1, &iArgType, szLex, &iLB, &iUB))
        ReportError (pibIn, RE_BADSTATE | RE_FATAL, szLex, NULL);

    if (!GetPunct (pibIn, szPunct, '=')) { /* no assignment */
        ReportError (pibIn, RE_LEXEXPECTED, "=", NULL);
    }

    if (iUB == -1) { /* scalar */
        /* check this is a declared state */
        if (GetVarType (pinfo->pvmGloVars, szLex) != ID_STATE)
            ReportError (pibIn, RE_BADSTATE | RE_FATAL, szLex, NULL);

        /* read assignment */
        GetStatement (pibIn, szEqn);
        UnrollEquation (pibIn, 0, szEqn, szEqnU);
        DefineVariable (pibIn, szLex, szEqnU, iKWCode);
    }
    else { /* array */
        /* read assignment */
        GetStatement (pibIn, szEqn);
        for (i = iLB; i < iUB; i++) {
            sprintf (szTmp, "%s_%ld", szLex, i); /* create names */
            /* check this is a declared state */
            if (GetVarType (pinfo->pvmGloVars, szTmp) != ID_STATE) {
                sprintf (szTmp, "%s[%ld]", szLex, i); /* recreate name */
                ReportError (pibIn, RE_BADSTATE | RE_FATAL, szTmp, NULL);
            }
            UnrollEquation (pibIn, i, szEqn, szEqnU);
            DefineVariable (pibIn, szTmp, szEqnU, iKWCode);
        }
    }

    if (!GetPunct (pibIn, szLex, CH_STMTTERM))
        ReportError (pibIn, RE_EXPECTED | RE_FATAL, ";", NULL);
}

/* -----
-----
```

```

    ProcessIdentifier
*/
void ProcessIdentifier (PINPUTBUF pibIn, PSTR szLex, PSTR szEqn, int
iKWCode)
{
    PSTRLEX szPunct, szTmp;
    PSTREQN szEqnU;
    PINPUTINFO pinfo;
    long i, iLB, iUB;

    pinfo = (PINPUTINFO) pibIn->pInfo;

    /* check that szLex is less than MAX_NAME characters */
    if ((i = strlen (szLex)) > MAX_NAME)
        ReportError (pibIn, RE_NAMETOOLONG | RE_FATAL, szLex, NULL);

    if (!GetPunct (pibIn, szPunct, '[')) { /* scalar */
        if (szPunct[0] == '=') { /* read assignment */
            GetStatement (pibIn, szEqn);
            UnrollEquation (pibIn, 0, szEqn, szEqnU);
            DefineVariable (pibIn, szLex, szEqnU, iKWCode);
            if (!GetPunct (pibIn, szLex, CH_STMTTERM))
                ReportError (pibIn, RE_EXPECTED | RE_FATAL, ";", NULL);
        }
        else if (szPunct[0] == CH_STMTTERM) { /* found a terminator */
            if (pinfo->wContext == CN_GLOBAL) {
                /* in the global section: assign 0; FB 13/6/99 */
                DefineVariable (pibIn, szLex, "0\0", iKWCode);
            }
            else
                ReportError (pibIn, RE_LEXEXPECTED | RE_FATAL, "= or [", NULL);
        }
        else /* nothing recognized: error */
            ReportError (pibIn, RE_LEXEXPECTED | RE_FATAL, "=, [ or ;", NULL);
    } /* end scalar */

    else { /* array */
        GetArrayBounds (pibIn, &iLB, &iUB);
        if (GetPunct (pibIn, szPunct, '=')) { /* read assignment */
            GetStatement (pibIn, szEqn);
            for (i = iLB; i < iUB; i++) {
                sprintf (szTmp, "%s_%ld", szLex, i); /* create names */
                UnrollEquation (pibIn, i, szEqn, szEqnU);
                DefineVariable (pibIn, szTmp, szEqnU, iKWCode);
            }
            if (!GetPunct (pibIn, szLex, CH_STMTTERM))
                ReportError (pibIn, RE_EXPECTED | RE_FATAL, ";", NULL);
        }
        else if (szPunct[0] == CH_STMTTERM) { /* found a terminator */
            if (pinfo->wContext == CN_GLOBAL) { /* in global section assign 0
*/
                for (i = iLB; i < iUB; i++) {
                    sprintf (szTmp, "%s_%ld", szLex, i); /* create names */

```

```

        DefineVariable (pibIn, szTmp, "0\0", iKWCode);
    }
}
else
    ReportError (pibIn, RE_LEXEXPECTED | RE_FATAL, "= or [", NULL);
}
else /* nothing recognized: error */
    ReportError (pibIn, RE_LEXEXPECTED | RE_FATAL, "= or ;", NULL);
} /* end array */

} /* ProcessIdentifier */

/* -----
-----
ProcessInlineStatement
*/
void ProcessInlineStatement (PINPUTBUF pibIn, PSTR szLex, PSTR szEqn,
                            int iKWCode)
{
    GetStatement (pibIn, szEqn);
    /* remove leading parenthesis */
    szEqn = szEqn + 1;
    /* remove ending parenthesis */
    szEqn[strlen(szEqn) - 1] = '\0';
    DefineVariable (pibIn, szLex, szEqn, iKWCode);

    if (!GetPunct (pibIn, szLex, CH_STMTTERM))
        ReportError (pibIn, RE_EXPECTED | RE_FATAL, ";", NULL);

} /* ProcessInlineStatement */

/* -----
-----
ProcessWord

    Processes the word szLex.
*/
void ProcessWord (PINPUTBUF pibIn, PSTR szLex, PSTR szEqn)
{
    int iErr = 0;
    int iKWCode, fContext;
    PSTRLEX szPunct;
    PINPUTINFO pinfo;

    static BOOL bCalcOutputsDefined = FALSE;
    static BOOL bDynamicsDefined = FALSE;
    static BOOL bInitializeDefined = FALSE;
    static BOOL bJacobianDefined = FALSE;

    if (!pibIn || !szLex || !szLex[0] || !szEqn)

```

```

    return;

pinfo = (PINPUTINFO) pibIn->pInfo;

iKWCode = GetKeywordCode (szLex, &fContext);

assert (pinfo->wContext != CN_END);

if ((pinfo->wContext == CN_END) || /* Beyond valid input ! */
    (iKWCode && /* Is a keyword */
     !(fContext & pinfo->wContext))) /* not in a valid context */
    ReportError (pibIn, RE_BADCONTEXT | RE_FATAL, szLex, NULL);

else {
    switch (iKWCode) {
        case KM_END:
            pinfo->wContext = CN_END;
            break;

        case KM_STATES:
        case KM_INPUTS:
        case KM_OUTPUTS:
        case KM_COMPARTMENTS:
            if (GetPunct (pibIn, szPunct, '=')) {
                if (GetPunct (pibIn, szPunct, CH_LBRACE))
                    GetVarList(pibIn, szLex, iKWCode);
                else
                    ReportError (pibIn, RE_EXPECTED | RE_FATAL, "{", NULL);
            }
            else
                ReportError (pibIn, RE_EXPECTED | RE_FATAL, "=", NULL);
            break;

        case KM_CALCOUPUTS:
            if (bCalcOutputsDefined)
                ReportError (pibIn, RE_DUPSECT | RE_FATAL, "CalcOutputs",
NULL);
            bCalcOutputsDefined = TRUE;
            if (!GetPunct (pibIn, szPunct, CH_LBRACE)) {
                szPunct[1] = CH_LBRACE;
                ReportError (pibIn, RE_EXPECTED | RE_FATAL, szPunct,
                    "* Section must be delimited by curly braces.");
            }
            else
                pinfo->wContext = KM_TO_CN(iKWCode);
            break;

        case KM_JACOB:
            if (bJacobianDefined)
                ReportError (pibIn, RE_DUPSECT | RE_FATAL, "Jacobian", NULL);
            bJacobianDefined = TRUE;
            if (!GetPunct (pibIn, szPunct, CH_LBRACE)) {
                szPunct[1] = CH_LBRACE;
                ReportError (pibIn, RE_EXPECTED | RE_FATAL, szPunct,

```

```

        /* Section must be delimited by curly braces.");
    }
else
    pinfo->wContext = KM_TO_CN(iKWCode);
break;

case KM_SCALE:
if (bInitializeDefined)
    ReportError (pibIn, RE_DUPSECT | RE_FATAL, "Initialize", NULL);
bInitializeDefined = TRUE;
if (!GetPunct (pibIn, szPunct, CH_LBRACE)) {
    szPunct[1] = CH_LBRACE;
    ReportError (pibIn, RE_EXPECTED | RE_FATAL, szPunct,
                  /* Section must be delimited by curly braces.");
}
else
    pinfo->wContext = KM_TO_CN(iKWCode);
break;

case KM_EVENTS:
case KM_ROOTS:
if (!GetPunct (pibIn, szPunct, CH_LBRACE)) {
    szPunct[1] = CH_LBRACE;
    ReportError (pibIn, RE_EXPECTED | RE_FATAL, szPunct,
                  /* Section must be delimited by curly braces.");
}
else
    pinfo->wContext = KM_TO_CN(iKWCode);
break;

case KM_DYNAMICS:
if (bDynamicsDefined)
    ReportError (pibIn, RE_DUPSECT | RE_FATAL, "Dynamics", NULL);
bDynamicsDefined = TRUE;

if (!GetPunct (pibIn, szPunct, CH_LBRACE)) {
    szPunct[1] = CH_LBRACE;
    ReportError (pibIn, RE_EXPECTED | RE_FATAL, szPunct,
                  /* Section must be delimited by curly braces.");
}
else
    pinfo->wContext = KM_TO_CN(iKWCode);
break;

case KM_DXDT: /* State equation definition */
    ProcessDTStatement (pibIn, szLex, szEqn, iKWCode);
break;

case KM_INLINE: /* Inline statement definition */
    ProcessInlineStatement (pibIn, szLex, szEqn, iKWCode);
break;

case KM_SBMLMODELS:
if (GetPunct (pibIn, szPunct, '=')) {

```

```

        if (GetPunct (pibIn, szPunct, CH_LBRACE)) {
            ReadSBMLModels (pibIn);
        }
        else
            ReportError (pibIn, RE_EXPECTED | RE_FATAL, "{", NULL);
    }
    else
        ReportError (pibIn, RE_EXPECTED | RE_FATAL, "=", NULL);
    break;

case KM_PKTEMPLATE:
    if (GetPunct (pibIn, szPunct, '=')) {
        if (GetPunct (pibIn, szPunct, CH_LBRACE)) {
            printf ("\nreading pharmacokinetic template ");
            ReadPKTemplate (pibIn);
        }
        else
            ReportError (pibIn, RE_EXPECTED | RE_FATAL, "{", NULL);
    }
    else
        ReportError (pibIn, RE_EXPECTED | RE_FATAL, "=", NULL);
    break;

default: /* Not a keyword, process identifier */
    ProcessIdentifier (pibIn, szLex, szEqn, iKWCode);
    break;
}

/* switch */

if (iErr)
    EatStatement (pibIn); /* Err in statement, eat to terminator */

} /* else */

} /* ProcessWord */

/*
-----
----- FindEnd

Returns 1 if the keyword "End" is found at the beginning of line,
eventually preceeded by white space, and zero otherwise.
*/
int FindEnd (PBUF pBuf, long N)
{
    char *c, *end;

    c = pBuf;
    end = pBuf + N;
    while (c < end) {
        // printf ("%c", *c);
        if (*c == CH_EOLN) { // eat up leading white space

```

```

        c++;
        while ((c < end) && (isspace(*c))) //(((*c == ' ') || (*c == '\t'))))
            c++;
        if (c < end) {
            // printf ("|%c", *c);
            if (((c+2) < end) && (*c == 'E') && (*(c+1) == 'n') && (*(c+2) ==
'd'))
                return (1);
            }
        c++;
    }

    // not found
    return(0);
}

} /* FindEnd */

/*
-----
----- ReadModel
Read the model definition in the given szFullPathname according to
the syntax described above and in the documentation.
*/
void ReadModel (PINPUTINFO pinfo, PINPUTINFO ptempinfo, PSTR szFileIn)
{
    INPUTBUF ibIn;
    PSTRLEX szLex; /* Lex elem of MAX_LEX length */
    PSTREQN szEqn; /* Equation buffer of MAX_EQN length */
    int iLexType;

    if (!InitBuffer (&ibIn, -1, szFileIn))
        ReportError (&ibIn, RE_INIT | RE_FATAL, "ReadModel", NULL);

    /* Attach info records to input buffer */
    ibIn.pInfo = (PVOID) pinfo;
    ibIn.pTempInfo = (PVOID) ptempinfo;

    /* immediately check whether a valid End is found */
    if (FindEnd(ibIn.pbufOrg, ibIn.lBufSize) == 0)
        ReportError (NULL, RE_NOEND | RE_FATAL, szFileIn, NULL);

    do { /* State machine for parsing syntax */
        NextLex (&ibIn, szLex, &iLexType);
        switch (iLexType) {
            case LX_NULL:
                pinfo->wContext = CN_END;
                break;

            case LX_IDENTIFIER:
                ProcessWord (&ibIn, szLex, szEqn);

```

```

        break;

case LX_PUNCT: case LX_EQNPUNCT:
    if (szLex[0] == CH_STMTTERM) {
        break;
    }
    else {
        if (szLex[0] == CH_RBRACE &&
            (pinfo->wContext & (CN_DYNAMICS | CN_JACOB | CN_SCALE))) {
            pinfo->wContext = CN_GLOBAL;
            break;
        }
        else {
            if (szLex[0] == CH_COMMENT) {
                SkipComment (&ibIn);
                break;
            }
            /* else: fall through! */
        }
    }
}

default:
    ReportError (&ibIn, RE_UNEXPECTED, szLex, "* Ignoring");
    break;

case LX_INTEGER:
case LX_FLOAT:
    ReportError (&ibIn, RE_UNEXPNUMBER, szLex, "* Ignoring");
    break;

} /* switch */

} while (pinfo->wContext != CN_END);

pinfo->wContext = CN_END;

// fclose (ibIn.pfileIn); already closed in InitBuffer

} /* ReadModel */

```